5. Actor model using the Akka framework

Nelma Moreira & **José Proença**

Concurrent programming (CC3040) 2023/2024

CISTER – U.Porto, Porto, Portugal

`https://fm-dcc.github.io/pc2324`

Overview

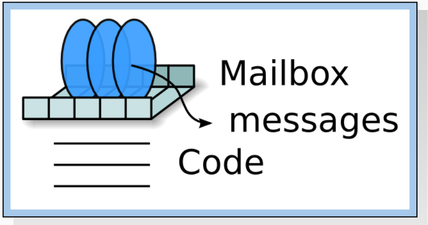
We are here

Blocks of sequential code running concurrently and sharing memory:

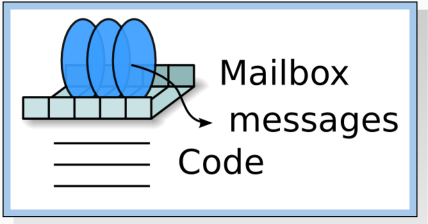
- What is Scala?
- Concurrency in Java and its memory model
- Basic concurrency blocks and libraries
- *Futures and Promises*
- *Data-Parallel Collections*
- *Reactive Programming (Concurrently)*
- *Software Transactional Memory*
- Actor model

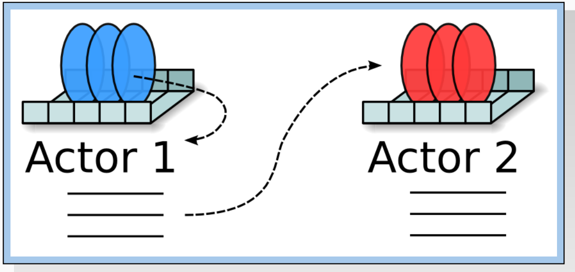
What is the actor model

- Asynchronous message exchange between actors
- Introduced in Erlang (we use Akka's actor library)



- Asynchronous message exchange between actors
- Introduced in Erlang (we use Akka's actor library)
- Active, autonomous, no shared memory, no synchronisation





What we will see

We will use the Akka framework for actors for:

- Declaring actor classes and creating actor instances
- Modelling actor state and complex actor behaviours
- Manipulating the actor hierarchy and the actor lifecycle
- The different message-passing patterns used in actor communication
- Error recovery using the built-in actor supervision mechanism
- Using remote actors to build concurrent and distributed programs

Documentation: `https://doc.akka.io/docs/akka`

Creating actors

Core concepts

Actor system

Hierarchical group of actors with shared configurations, supporting actor creation and logging.

Actor class

Template that describes the states and behaviour of an actor, used to create instances.

Actor instance

Entity that exists at runtime, with a state and capable of sending and receiving messages.

Mailbox

Memory block that is used to buffer messages for a given actor instance.

Actor reference

Object that allows an object to send messages to a specific actor instance.

Dispatcher

Component that decides when actors are allowed to process messages. In Akka every dispatcher is also an execution context.

My first actor (class) in Akka

```scala
import akka.actor._
import akka.event.Logging

class HelloActor(val hello: String)
    extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case `hello` =>
      log.info(
        s"Received a '$hello'... $hello!")
    case msg     =>
      log.info(
        s"Unexpected message '$msg'")
      context.stop(self)
  }
}
```

- Each `HelloActor` receives messages
- ... if it receives its `hello`, it logs and continues
- ... if it receives something else, it stops
- `context` – provides core functions, such as `stop`
- `self` – is the instance's actor reference

Configuring an actor in Akka

```scala
object HelloActor { // companion
  // two factory methods below
  def props(hello: String) =
    Props(new HelloActor(hello))
  def propsAlt(hello: String) =
    Props(classOf[HelloActor], hello)
  //def propsAlt2 = Props[HelloActor]
}
```

Actor configuration

- actor class
- constructor arguments
- mailbox
- dispatcher

Props

- can receive a block of code, used each time a new actor instance is created;
- can receive a `Class` object and its arguments
- can be sent over the network (should be self-contained)
- avoid creating `Props` in the actor class, and use factory methods instead

My first actor system with an instance

```scala
// in build.sbt:
libraryDependencies ++= Seq( ...
  ,"com.typesafe.akka" %% "akka-actor" % "2.8.5"
  ,"com.typesafe.akka" %% "akka-remote" % "2.8.5"
)
```

```scala
lazy val ourSystem = akka.actor.ActorSystem("OurExampleSystem")
```

```scala
object ActorsCreate extends App {
  val hiActor: ActorRef =
    ourSystem.actorOf(HelloActor.props("ola"), name = "greeter")
  hiActor ! "ola"
  Thread.sleep(1000)
  hiActor ! "hi"
  Thread.sleep(1000)
  ourSystem.terminate()
}
```

Unhandled messages?

```
class DeafActor extends Actor {
  val log = Logging(context.system, this)
  def receive = PartialFunction.empty
  override def unhandled(msg: Any) = msg match {
    case msg: String => log.info(s"I␣do␣not␣hear␣'$msg'")
    case msg         => super.unhandled(msg)
  }
}
```

```
object ActorsUnhandled extends App {
  val deafActor: ActorRef =
    ourSystem.actorOf(Props[DeafActor], name = "deafy")
  deafActor ! "ola"
  Thread.sleep(1000)
  deafActor ! 1234
  Thread.sleep(1000)
  ourSystem.terminate()
}
```

Modelling actor behaviour

My 2nd example in Akka: a (stateful) countdown

```
class CountdownActor extends Actor{
  var n = 10
              // never do this
  def receive = if (n > 0) {
    case "count" =>
      log(s"n␣=␣$n")
      n -= 1
  } else PartialFunction.empty
}
```

My 2nd example in Akka: a (stateful) countdown

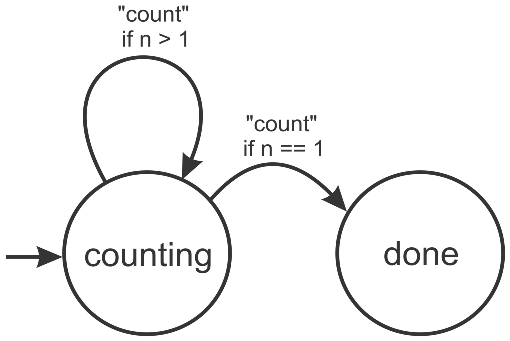
Not allowed in Akka:

```
class CountdownActor extends Actor{
  var n = 10
                // never do this
  def receive = if (n > 0) {
    case "count" =>
      log(s"n␣=␣$n")
      n -= 1
  } else PartialFunction.empty
}
```

Correct in Akka, using `become`:

```
class CountdownActor extends Actor {
  val log = Logging(context.system,
        this)
  var n = 10
  def counting: Actor.Receive = {
    case "count" =>
      n -= 1
      log.info(s"n␣=␣$n")
      if (n == 0) context.become(done)
  }
  def done = PartialFunction.empty
  def receive = counting
}
```

Actor as a transition system



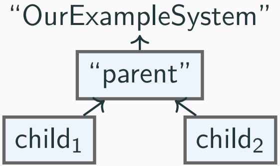
[in *"Learning Concurrent Programming in Scala"*, pg. 278]

Running the countdown

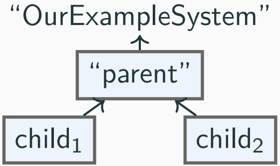
```
object ActorsCountdown extends App {
  val countdown = ourSystem.actorOf(Props[CountdownActor])
  for (i <- 0 until 20) countdown ! "count"
  Thread.sleep(1000)
  ourSystem.terminate()
}
```

Actor hierarchy and lifecycle

New example with a parent



New example with a parent

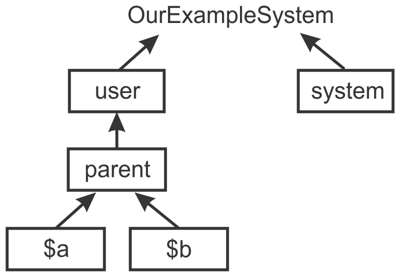



```
class ChildActor extends Actor {
  val log =
      Logging(context.system, this)
  def receive = {
    case "sayhi" =>
      val parent = context.parent
      log.info(s"my␣parent␣$parent␣
          made␣me␣say␣hi!")
  }
  override def postStop() {
    log.info("child␣stopped!")
  }
}
```

```
class ParentActor extends Actor {
  val log = Logging(context.system,
      this)
  def receive = {
    case "create" =>
      context.actorOf(Props[ChildActor])
      log.info(s"created␣a␣kid;␣
          children␣=␣
          ${context.children}")
    case "sayhi" =>
      log.info("Kids,␣say␣hi!")
      for (c <- context.children) c !
          "sayhi"
    case "stop" =>
      log.info("parent␣stopping")
      context.stop(self)
  }
}
```

A more complete view of the hierarchy

```
object ActorsHierarchy extends App {
  val parent =
      ourSystem.actorOf(Props[ParentActor],
      "parent")
  parent ! "create"
  parent ! "create"
  Thread.sleep(1000)
  parent ! "sayhi"
  Thread.sleep(1000)
  parent ! "stop"
  Thread.sleep(1000)
  ourSystem.terminate()
}
```



in *"Learning Concurrent Programming in Scala"*, pg. 284

- ActorSystem
- sys.terminate
- sys/ctxt.actorOf

- ctxt.stop
- ctxt.become
- ctxt.children
- ctxt.parent

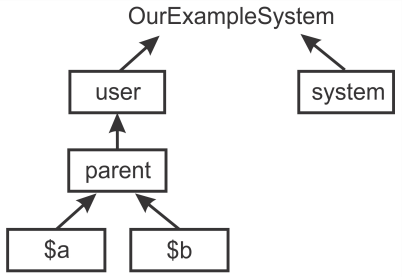
A more complete view of the hierarchy

- parent actor stops $\Rightarrow$ its children stop
- user and system:

  are guardian actors – at the top of the hierarchy, to log, restart actors, etc.
- hierarchy visible when printing an actor ref, e.g., for the first child;

  `akka://OurExampleSystem/user/parent/$a`

A more complete view of the hierarchy



in *"Learning Concurrent Programming in Scala"*, pg. 284

- parent actor stops ⇒ its children stop
- user and system:

  are guardian actors – at the top of the hierarchy, to log, restart actors, etc.
- hierarchy visible when printing an actor ref, e.g., for the first child;

  `akka://OurExampleSystem/user/parent/$a`
- Next: `ctxt.actorSelection(path)`

- ActorSystem
- sys.terminate
- sys/ctxt.actorOf

- ctxt.stop
- ctxt.become
- ctxt.children
- ctxt.parent

Discovering actors in the hierarchy

```
class CheckActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case path: String =>
      log.info(s"checking path $path")
      context.actorSelection(path) ! Identify(path)
    case ActorIdentity(path, Some(ref)) =>
      log.info(s"found actor $ref at $path")
    case ActorIdentity(path, None) =>
      log.info(s"could not find an actor at $path")
  }
}
```

Discovering actors in the hierarchy

```
class CheckActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case path: String =>
      log.info(s"checking␣path␣$path")
      context.actorSelection(path) ! Identify(path)
    case ActorIdentity(path, Some(ref)) =>
      log.info(s"found␣actor␣$ref␣at␣$path")
    case ActorIdentity(path, None) =>
      log.info(s"could␣not␣find␣an␣actor␣at␣$path")
  }
}
```

```
val checker = ourSystem.actorOf(Props[CheckActor], "checker")

checker ! "../*"      // finds the checker and its siblings
checker ! "../../*"   // finds user and system guardians
checker ! "/system/*" // finds internal actors
checker ! "/user/checker2" // logs that no actors were found
```

Once an actor throws an exception...

When an actor throws an exception, a new “replacement” actor is created, with the same:

- arguments
- mailbox
- ActorRef

Once an actor throws an exception...

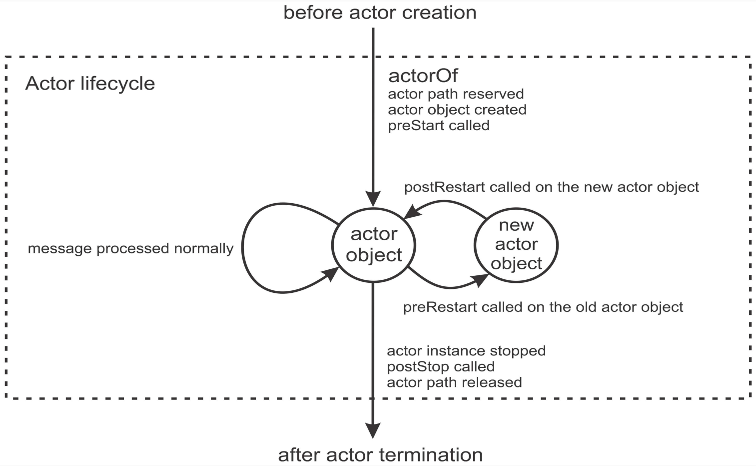
When an actor throws an exception, a new “replacement” actor is created, with the same:

- arguments
- mailbox
- ActorRef
    - hence never leak the actual this reference!

Actor lifecycle

- Created – `actorOf`
- Before starting to process messages – `preStart()`
- After an exception – `preRestart(t: Throwable, msg: Option[Any])`
  - before creating a new actor
  - when all children are stopped
- After recreating a restarted actor – `postRestart(t: Throwable)`
  - the new actor is then assigned the previous mailbox
- After an actor terminates – `postStop()`
  - called by the default implementation of `preRestart`

Actor lifecycle in a diagram



[in *"Learning Concurrent Programming in Scala"*, pg. 289]

Synchrony vs. Asynchrony

Sending x and y from A to B

Synchronous (as in CCS)

$A = x! \,.\, y!$

$B = x? \,.\, y?$

$A \mid B \backslash \{x, y\}$

Sending x and y from A to B

Synchronous (as in CCS)

$A = x!\,.\,y!$

$B = x?\,.\,y?$

$A \mid B \backslash \{x, y\}$

$\Rightarrow \quad \tau_x . \tau_y$

Sending x and y from A to B

Synchronous (as in CCS)

$A = x! \,.\, y!$

$B = x? \,.\, y?$

$A \mid B \backslash \{x, y\}$

$\Rightarrow \quad \tau_x . \tau_y$

Asynchronous (as in Akka)

$x!$ happens before $y!$

$x?$ happens before $y?$

Sending x and y from A to B

Synchronous (as in CCS)

$A = x! \,.\, y!$

$B = x? \,.\, y?$

$A \mid B \backslash \{x, y\}$

$\Rightarrow \qquad \tau_x . \tau_y$

Asynchronous (as in Akka)

$x!$ happens before $y!$

$x?$ happens before $y?$

$x!$ happens before $x?$

$y!$ happens before $y?$

Sending x and y from A to B

Synchronous (as in CCS)

$A = x!\,.\,y!$

$B = x?\,.\,y?$

$A \mid B\backslash\{x, y\}$

$\Rightarrow \quad \tau_x.\tau_y$

Asynchronous (as in Akka)

$x!$ happens before $y!$

$x?$ happens before $y?$

$x!$ happens before $x?$

$y!$ happens before $y?$

$y!$?? $x?$

Sending x and y from A to B

Synchronous (as in CCS)

$A = x! \,.\, y!$

$B = x? \,.\, y?$

$A \mid B \backslash \{x, y\}$

$\Rightarrow \quad \tau_x . \tau_y$

Different formalisations for global beh.:

- Message sequence charts
- Event structures
- Automata over interactions
- Choreographies:

  $A \to B : x \;\; ; \;\; A \to B : y$

Asynchronous (as in Akka)

$x!$ happens before $y!$

$x?$ happens before $y?$

$x!$ happens before $x?$

$y!$ happens before $y?$

$y!$?? $x?$

No duplication

No messages lost

No messages reordered

No blocking send

Synchrony modelled with Asynchrony?

and vice-versa?

Diamond problem: sending by two routes

$A \rightarrow B : x;$
$A \rightarrow C : y;$
$C \rightarrow B : z$

B must be ready to receive ‘x?’ and ‘z?’ by any order

Error recovery with actors

Stopping an actor

Main ways to stop an actor:

- `context.stop` – stops all actors, once they finish processing their current message
- `Kill` message – stops the target actor once it is received
- `PoisonPill` message – stops the target actor after processing all the messages currently in its mailbox

Stopping an actor

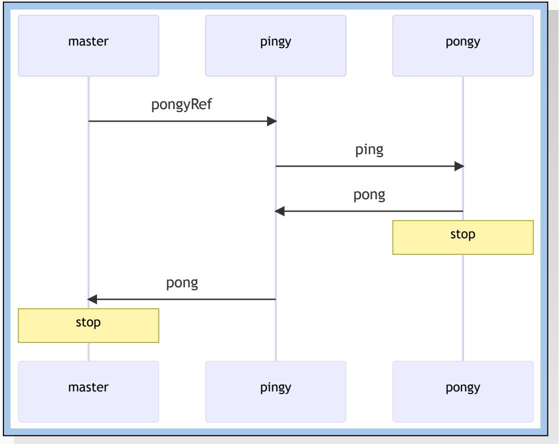
Main ways to stop an actor:

- `context.stop` – stops all actors, once they finish processing their current message
- `Kill` message – stops the target actor once it is received
- `PoisonPill` message – stops the target actor after processing all the messages currently in its mailbox

Stopping in more complex scenarios:

- Using Akka's DeathWatch (next slide)

Pingy-Pongy example



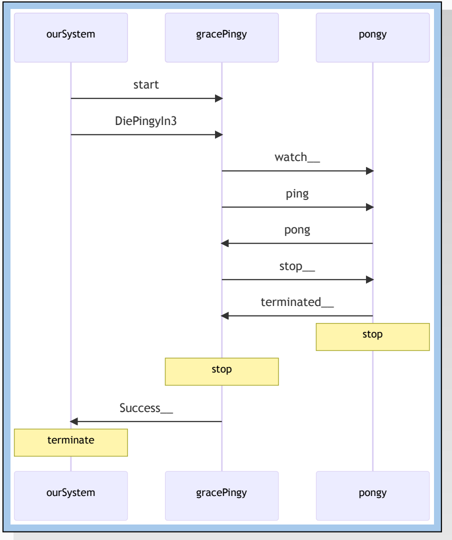
- Example used in the book to illustrate the ask-reply pattern
- (in pingy: `val reply = pongy ? "ping"`)
- We will adapt it for a graceful shutdown

Graceful Pingy-Pongy

```
class GracefulPingy extends Actor {
  val log = Logging(context.system, this)
  val pongy =
    context.actorOf(Props[Pongy], "pongy")
  context.watch(pongy)
  def receive = {
    case "start" => pongy ! "ping"
    case "pong"  => log.info("Got␣a␣pong")
    case "Die,␣Pingy!" =>
      context.stop(pongy)
    case Terminated(‘pongy‘) =>
      context.stop(self)
} }
```

```
class Pongy extends Actor {
  val log =
    Logging(context.system,this)
  def receive = {
    case "ping" =>
      log.info("Got␣a␣ping␣--␣
          ponging␣back!")
      sender ! "pong"
  }
  override def postStop() =
      log.info("pongy␣going␣
      down")
}
```

Running the gracefull app



Mechanism 1 (pingy $\leftrightarrow$ pongy)

- `context.watch(pongy)` – the `DeathWatch`
- wait for `Terminated` message

Mechanism 2 (ourSystem $\leftrightarrow$ pingy)

- ask to `"Die"`
- check if it terminated – using Futures

Running the gracefull app (code)

```
import akka.pattern.gracefulStop

object CommunicatingGracefulStop extends App {
  val gracePingy = ourSystem.actorOf(Props[GracefulPingy], "gracePingy")
  gracePingy ! "start"

  val stopped = gracefulStop(gracePingy, 3.seconds, "Die,␣Pingy!")
  stopped onComplete { // stopped is a Future (not covered)
    case Success(x) =>
      log("graceful␣shutdown␣successful")
      ourSystem.terminate()
    case Failure(t) =>
      log("grace␣not␣stopped!")
      ourSystem.terminate()
} }
```

Handling children’s exceptions (Actor supervision)

```
class Naughty extends Actor {
  val log = Logging(context.system,this)
  def receive = {
    case s: String => log.info(s)
    case msg => throw new
        RuntimeException
  }
  override def postRestart(t:Throwable)=
    log.info("naughty␣restarted")
}
```

```
import SupervisorStrategy._
class Supervisor extends Actor {
  context.actorOf(Props[Naughty],
      "naughty")
  def receive = PartialFunction.empty
  override val supervisorStrategy =
    OneForOneStrategy() {
      case ake: ActorKilledException
          => Restart
      case _ => Escalate
} }
```

```
ourSystem.actorOf(Props[Supervisor], "super")
val children = ourSystem.actorSelection("/user/super/*")
children ! "hello" // succeeds
children ! Kill    // stops naughty, but super restarts it
children ! "sorry␣about␣that" // succeeds
children ! "kaboom".toList    // naughty and super throw exception
```

Remote actors over TCP

Compilation with remote actors

build.sbt
needs to import akka-remote:

```
libraryDependencies ++= Seq(
   ...
  ,"com.typesafe.akka" %% "akka-actor" % "2.8.5" // or older
  ,"com.typesafe.akka" %% "akka-remote" % "2.8.5"
)
```

Network configured with Netty library

```
import com.typesafe.config._
def remotingConfig(port: Int) = ConfigFactory.parseString(s"""
  akka {
    actor.provider = "akka.remote.RemoteActorRefProvider"
    remote {
      enabled-transports = ["akka.remote.netty.tcp"]
      netty.tcp {
        hostname = "127.0.0.1"
        port = $port }
    }
  }""")
def remotingSystem(name: String, port: Int): ActorSystem =
     ActorSystem(name, remotingConfig(port))
```

Remote Pingy-Pongy – running two Apps!

```scala
object RemotingPongySystem extends App {
  val system =
       remotingSystem("PongyDimension",
       24321)
  val pongy = system.actorOf(Props[Pongy],
       "pongy")
  Thread.sleep(15000)
  system.terminate()
}
```

```scala
object RemotingPingySystem extends App {
  val system =
       remotingSystem("PingyDimension",
       24567)
  val runner = system.actorOf(Props[Runner],
       "runner")
  runner ! "start"
  Thread.sleep(5000)
  system.terminate()
}
```

```scala
class Runner extends Actor {
  val log = Logging(context.system, this)
  val pingy = context.actorOf(Props[Pingy], "pingy")
  def receive = {
    case "start" =>
      val pongySys =
           "akka.tcp://PongyDimension@127.0.0.1:24321"
      val pongyPath = "/user/pongy"
      val url = pongySys + pongyPath
      val selection = context.actorSelection(url)
      selection ! Identify(0)
    case ActorIdentity(0, Some(ref)) =>
      pingy ! ref
    case ActorIdentity(0, None) =>
      log.info("Something's␣wrong␣-␣ain't␣no␣pongy␣
           anywhere!")
      context.stop(self)
    case "pong" =>
      log.info("got␣a␣pong␣from␣another␣dimension.")
      context.stop(self)
  }
}
```

Running the multi-dimensional Pingy-Pongy

- Start the `RemotingPongySystem`
- Start the `RemotingPingySystem` within 15 sec.
- Use different SBT instances
- Runner in `PingyDimension` should get a “pong” soon

Running the multi-dimensional Pingy-Pongy

- Start the `RemotingPongySystem`
- Start the `RemotingPingySystem` within 15 sec.
- Use different SBT instances
- Runner in `PingyDimension` should get a “pong” soon

Deployment logic vs. Application logic

- Deployment log.: setting up network communication
- Application log.: interactions between agents
- These should be kept in separate
- In our example, `Runner` handles deployment logic

Wrapping up remote actors

Steps for handling remote actors

- Declaring each actor system with appropriate remoting configuration
- Starting each actor system in separate processes or on separate machines
- Obtain actor references by using actor path selection
- Transparently send messages by using these actor references

Wrapping up Actors

- Declare actor classes and create actor instances
- Model actor state and complex actor behaviours
- Manipulate the actor hierarchy and the actor lifecycle
- Use some message-passing patterns used in actor communication
- Use error recovery with the built-in actor supervision mechanism
- Use remote actors to build concurrent and distributed programs

Documentation: `https://doc.akka.io/docs/akka`